Limits and restrictions

The following modules allow you to regulate access to the documents of your websites—require users to authenticate, match a set of rules, or simply restrict the access to certain visitors.

Auth_basic module

The `auth_basic` module enables the basic authentication functionality. With the two directives that it brings forth, you can make it such that a specific location of your website (or your server) is restricted to users who authenticate with a username and password:

```
location /admin/ {
    auth_basic "Admin control panel"; # variables are supported
    auth_basic_user_file access/password_file;
}
```

The first directive, `auth_basic`, can be set to either `off` or a text message, usually referred to as *authentication challenge* or *authentication realm*. This message is displayed by the web browsers in a username/password box when a client attempts to access the protected resource.

The second one, `auth_basic_user_file`, defines the path of the password file relative to the directory of the configuration file. A password file is formed of lines respecting the following syntax: `username:[{SCHEME}]password[:comment]`. Where:

- `username`: a plain text user name
- `{SCHEME}`: optionally, the password hashing method. There are currently three supported schemes: `{PLAIN}` for plain text passwords, `{SHA}` for SHA-1 hashing, and `{SSHA}` for salted SHA-1 hashing.
- `password`: the password
- `comment`: a plain text comment for your own use

If you fail to specify a scheme, the password will need to be encrypted with the `crypt(3)` function, for example with the help of the `htpasswd` command-line utility from the Apache packages.

If you aren't too keen on installing Apache on your system just for the sake of the `htpasswd` tool, you may resort to online tools, as there are plenty of them available. Fire up your favorite search engine and type `online htpasswd`.

Access

Two important directives are brought up by this module: `allow` and `deny`. They let you allow or deny access to a resource for a specific IP address or IP address range.

Both directives have the same syntax: `allow IP | CIDR | unix: | all`, where `IP` is an IP address, `CIDR` is an IP address range (CIDR syntax), unix: represents all UNIX domain sockets, and `all` specifies that the directive applies to all clients:

```
location {
    allow 127.0.0.1; # allow local IP address
    allow unix:; # allow UNIX domain sockets
    deny all; # deny all other IP addresses
}
```

Note that rules are processed from top-down—if your first instruction is `deny all`, all possible `allow` exceptions that you place afterwards will have no effect. The opposite is also true—if you start with `allow all`, all possible `deny` directives that you place afterwards will have no effect, as you already allowed all the IP addresses.

Limit connections

The mechanism induced by this module is a little more complex than the regular ones. It allows you to define the maximum number of simultaneous connections to the server for a specific *zone*.

The first step is to define the zone using the `limit_conn_zone` directive:

- Directive syntax: `limit_conn_zone $variable zone=name:size;`
- `$variable` is the variable that will be used to differentiate one client from another, typically `$binary_remote_addr`—the IP address of the client in the binary format (this is more efficient than ASCII)
- `name` is an arbitrary name given to the zone
- `size` is the maximum size you allocate to the table storing session states

The following example defines the zones based on the client IP addresses:

```
limit_conn_zone $binary_remote_addr zone=myzone:10m;
```

Now that you have defined a zone, you may limit the connections using `limit_conn`:

```
limit_conn zone_name connection_limit;
```

When applied to the previous example, it becomes:

```
location /downloads/ {
    limit_conn myzone 1;
}
```

As a result, requests that share the same `$binary_remote_addr` are subject to the connection limit (one simultaneous connection). If the limit is reached, all additional concurrent requests will be answered with a `503 Service unavailable` HTTP response. This response code can be overridden if you specify another code via the `limit_conn_status` directive. If you wish to log client requests that are affected by the limits you have set, enable the `limit_conn_log_level` directive, and specify the log level (`info | notice | warn | error`).

Limit request

In a similar fashion, the *Limit request* module allows you to limit the number of requests for a defined zone.

Defining the zone is done via the `limit_req_zone` directive; its syntax differs from the *Limit zone* equivalent directive:

```
limit_req_zone $variable zone=name:max_memory_size rate=rate;
```

The directive parameters are identical except for the trailing `rate`: expressed in requests per second (r/s) or requests per minute (r/m). It defines a request rate that will be applied to clients where the zone is enabled. To apply a zone to a location, use the `limit_req` directive:

```
limit_req zone=name burst=burst [nodelay];
```

The `burst` parameter defines the maximum possible bursts of requests—when the amount of requests received from a client exceeds the limit defined in the zone, the responses are delayed in a manner that respects the rate that you defined. To a certain extent, only a maximum of `burst` requests will be accepted simultaneously. Past this limit, Nginx returns a `503 Service Unavailable` HTTP error response. This response code can be overridden if you specify another code via the `limit_req_status` directive.

```
limit_req_zone $binary_remote_addr zone=myzone:10m rate=2r/s;
[…]
location /downloads/ {
  limit_req zone=myzone burst=10;
  limit_req_status 404; # returns a 403 error if limit is exceeded
}
```

If you wish to log client requests that are affected by the limits you have set, enable the `limit_req_log_level` directive, and specify the log level (`info | notice | warn | error`).

Auth_request

The **auth_request** module was implemented in the recent versions of Nginx, and allows you to allow or deny access to a resource based on the result of a sub-request. Nginx calls the URI that you specify via the `auth_request` directive: if the sub-request returns a 2XX response code (that is, `HTTP/200 OK`), access is allowed. If the sub-request returns a 401 or 403 status code, access is denied, and Nginx forwards the response code to the client. Should the backend return any other response code, Nginx will consider it to be an error and deny access to the resource.

```
location /downloads/ {
    # if the script below returns a 200 status code,
    # the download is authorized
    auth_request /authorization.php;
}
```

Additionally, the module offers a second directive called `auth_request_set`, allowing you to set a variable after the sub-request is executed. You can insert variables that originate from the sub-request upstream (`$upstream_http_*`) such as `$upstream_http_server` or other HTTP headers from the server response.

```
location /downloads/ {
    # requests authorization from PHP script
    auth_request /authorization.php;
    # assuming authorization is granted, get filename from
    # sub-request response header and redirect
    auth_request_set $filename "${upstream_http_x_filename}.zip";
    rewrite ^ /documents/$filename;
}
```

Content and encoding

The following set of modules provides functionalities having an effect on the contents served to the client, either by modifying the way the response is encoded, by affecting the headers, or by generating a response from scratch.